Practical (?) Applications of Reflection

Jackie Kay
C++ London
July 4th, 2017
jackieokay.com

"Narcissus", Caravaggio, circa 1597–1599

Alternative title image: "Metamorphosis of Narcissus", Salvador Dalí, 1937

A problem

Code duplication due to hardcoding knowledge about the layout of a type.

```
struct Box    { Point pos; float length; float width; };
struct Circle { Point pos; float radius; };

// Primitive shapes defined by rendering library
void draw(const Box&);
void draw(const Circle&);

template<typename T>
void draw(const T& shape) {
  // We know T is composed of only boxes and circles.
  // But how do we know how to access them?
}
```

One solution is costly at runtime:

```
struct CompositeShape {
  std::vector<Box> boxes;
  std::vector<Circle> circles;
};

void draw(const CompositeShape& shape) {
  for (const auto& box : shape.boxes) {
    draw(box);
  }
  for (const auto& circle : shape.circle) {
    draw(circle);
  }
}
```

Another solution is costly for programmer time:

```
struct Capsule {
  std::array<Circle, 2> ends;
  Box middle;
};

void draw(const Capsule& shape) {
  for (const auto& circle : shape.ends) {
    draw(circle);
  }
  draw(shape.middle);
}

// Imagine writing this for hundreds of objects
// Now imagine maintaining it
```

How about an API that gets all the members for a generic type?

```
// Ideal syntax

template<typename T>
void draw(const T& shape) {
  for... (auto& member : reflect(shape).members()) {
    if constexpr (member.drawable()) {
      draw(member);
    }
  }
}
```

Let's call this "introspection".

C++ solutions

- Schema-based code generation
    - Cap'n Proto
- Compiler tooling
    - siplasplas
- "Adapt struct" macros
    - Fusion, Hana
- Arcane secrets
    - POD Flat Reflection (formerly magic_get)

Cap'n Proto

C++ code is generated from a schema:

```
struct Person {
  name @0 :Text;
  age  @1 :UInt32;
}
```

becomes

```
// name @1 :Text;
::capnp::Text::Reader getName();
// age @0 :Int32;
int32_t getAge();
```

siplasplas

Generate compile-time reflection info with libclang

```
class Foo { int i; };
// generates in a separate header:
template<>
class Class<Foo> {
public:
    using Fields = typelist<Field<SourceInfo<
                string<'i'>,                                 // name
                string<'f', 'o', 'o', '.', 'h'>, // file
                4                                            // line
            >,
            decltype(&Foo::i), &Foo::i, // Pointer
        >>;
};
```

Adapt struct macros

```
// from Boost Hana documentation, boostorg.github.io/hana
struct Person {
  BOOST_HANA_DEFINE_STRUCT(Person,
    (std::string, name),
    (std::string, last_name),
    (int, age));
};
// or, if Person is externally defined
BOOST_HANA_ADAPT_STRUCT(Person, name, last_name, age);

Person presenter{"Jackie", "Kay", 24};
hana::for_each(presenter, [](auto pair) {
  std::cout << hana::to<char const*>(hana::first(pair))
            << ": " << hana::second(pair) << std::endl;
});
```

How does it work?

- For each member, the macro:
- Generates generic set and get functions
- Stringizes the field name into constexpr string
- Uses the identifier name and type to get the member pointer
- Associates this into tuple of constexpr string, member pointer accessor pairs.
- Template specialization required for 1 member, 2 members, ... up to N members

POD Flat Reflection (pfr)

```
template<size_t I, typename T>
void print_members_recursive(const T& t) {
  if constexpr (I == pfr::tuple_size<T>{}) {
    return;
  } else {
    std::cout << pfr::flat_get<I>(t) << "\n";
    print_members_recursive<I + 1>(t);
  }
}
struct Person {
  const char* name;
  const char* last_name;
  int age;
};
Person presenter{"Jackie", "Kay", 24};
print_members_recursive<0>(presenter);
```

How does it work?

- C++17 version uses structured bindings and no macros
- That sounds much better... right?

```cpp
// full namespaces and noexcept omitted for brevity
template <class T>
constexpr auto as_tuple_impl(T&& /*val*/, size_t_<0>) {
  return sequence_tuple::tuple<>{};
}

template <class T>
constexpr auto as_tuple_impl(T&& val, size_t_<1>) {
  auto& [a] = std::forward<T>(val);
  return detail::make_tuple_of_references(a);
}

template <class T>
constexpr auto as_tuple_impl(T&& val, size_t_<2>) {
  auto& [a,b] = std::forward<T>(val);
  return detail::make_tuple_of_references(a,b);
}

// etc...
```

```
template <class T>
constexpr auto as_tuple_impl(T&& val, size_t_<100>) {
  auto& [
    a,b,c,d,e,f,g,h,j,k,l,m,n,p,q,r,s,t,u,v,w,x,y,z,A,B,C,
    D,E,F,G,H,J,K,L,M,N,P,Q,R,S,T,U,V,W,X,Y,Z,aa,ab,ac,ad,
    ae,af,ag,ah,aj,ak,al,am,an,ap,aq,ar,as,at,au,av,aw,ax,
    ay,az,aA,aB,aC,aD,aE,aF,aG,aH,aJ,aK,aL,aM,aN,aP,aQ,aR,
    aS,aT,aU,aV,aW,aX,aY,aZ,ba,bb,bc,bd
  ] = std::forward<T>(val);

  return detail::make_tuple_of_references(
    a,b,c,d,e,f,g,h,j,k,l,m,n,p,q,r,s,t,u,v,w,x,y,z,A,B,C,
    D,E,F,G,H,J,K,L,M,N,P,Q,R,S,T,U,V,W,X,Y,Z,aa,ab,ac,ad,
    ae,af,ag,ah,aj,ak,al,am,an,ap,aq,ar,as,at,au,av,aw,ax,
    ay,az,aA,aB,aC,aD,aE,aF,aG,aH,aJ,aK,aL,aM,aN,aP,aQ,aR,
    aS,aT,aU,aV,aW,aX,aY,aZ,ba,bb,bc,bd
  );
}
```

Other languages

- Python: Flexible and powerful, user-friendly syntax
- Java: Extensive runtime API
- C#: Runtime introspection and synthesis API
- D: powerful compile-time intrsopection, mixins, hygienic macros

Proposed solutions for C++

reflexpr

- P0194R3 by Matúš Chochlík and Axel Naumann
- I recommend P0578R0, "Static Reflection in a Nutshell" by Chochlík, Naumann and David Sankel
- Clang implementation and documentation

reflexpr: raw API

```
namespace meta = std::meta;
using MetaInfo = reflexpr(Person);
std::cout << "A " << meta::get_base_name_v<MetaInfo>
          << " is made up of "
          << meta::get_size<MetaInfo> << " things.\n";
```

operator$/cpp3k

- P0590R0 by Andrew Sutton and Herb Sutter
- Clang implementation also available
- Choice of $ is controversional: supported as valid identifier as an extension in most compilers
  - But there's more to it than just the symbol

operator$

```
namespace meta = cpp3k::meta::v1;
constexpr auto info = $Person;
std::cout << "A " << info.name() << " is made up of "
          << info.member_variables().size()
          << " things.\n";
```

Which API is better?

Well, let's figure out how we want to use it first!

Warm-up: equality operators

```
template<typename T>
bool equal(const T& a, const T& b) {
  if constexpr (equality_comparable<T>()) {
    return a == b;
  } else if constexpr (iterable<T>()) {
    if (a.size() != b.size()) {
      return false;
    }
    for (int i = 0; i < a.size(); ++i) {
      if (!equal(a[i], b[i])) return false;
    }
    return true;
  } else { /* Time for reflection */ }
}
```

With reflexpr

```
using MetaT = reflexpr(T);
static_assert(meta::Record<MetaT>,
  "Reached non-equality comparable leaf member.");
bool result = true;
meta::for_each<meta::get_data_members_m<MetaT>>(
  [&a, &b, &result](auto&& member) {
    using M = typename std::decay_t<decltype(member)>;
    constexpr auto p = meta::get_pointer<M>::value;
    result = result && equal(a.*p, b.*p);
  }
);
return result;
```

With cpp3k

```
static_assert(refl::is_member_type<T>(),
  "Reached non-equality comparable leaf member.");
bool result = true;
meta::for_each($T.member_variables(),
  [&a, &b, &result](auto&& member){
    constexpr auto p = member.pointer();
    result = result && equal(a.*p, b.*p);
  }
);
return result;
```

Fold expressions

reflexpr can express sequences as parameter packs:

```
template<typename ...Pack>
struct compare_fold {
  static constexpr auto apply(const T& a, const T& b) {
    return (equal(a.*pointer<Pack>(), b.*pointer<Pack>())
            && ...);
  }
};
// ...
meta::unpack_sequence_t<meta::get_data_members_m<MetaT>,
    compare_fold>::apply(a, b);
```

Can probably optimize better than for_each

Serialization/deserialization

JSON deserialization

Louis Dionne's Meeting C++ 2016 keynote showed a JSON serializer using a value-semantics metaprogramming mini-library built on top of `reflexpr`.

How about deserialization?

```
template<typename T>
auto deserialize(std::string_view& src, T& dst);
```

Requires matching a runtime string to a metainfo.

Linear matching

```
// input: string_view representing the string to parse
// parsed_keys: all key strings for a JSON object
// result: error code (gets returned out of deserialize)
for (const auto& key : parsed_keys) {
  const auto& value = parse_next_value(input);
  meta::for_each($T.member_variables(),
    [&dst, &key, &value, &result](auto&& m) {
      if (key == m.name()) {
        if (result = deserialize(value, dst.*m.pointer());
            result != deserialize_result::success) {
          return;
        }
      }
    }
  );
}
```

Observations

- Same pattern of recursively applying an operation over all members of a struct as the equality operator example.
- String matching could be costly at runtime: if T has n members, O(n) string comparisons per JSON key.

Program options

Program options: interface

Struct member name corresponds to command line flag and abbreviation.

```
struct ProgramOptions {
  std::string filename;  // --filename, -f
  int iterations = 100;  // --iterations, -i
  bool help = false;     // --help, -h
  float foo;             // --foo, -o
};

// Returns nullopt if there was a parsing error
template<typename T>
optional<T> parse(int argc, char** argv const);
```

Program options: outline

- Accumulate a Hana compile-time map with pairs: ("--member_name", metainfo)
- Take 'c', the first character in "member_name". If "-c" not in the map, add ("-c", "metainfo"), else check next character.
- `parse` iterates over `argv` and converts a runtime `const char*` to constexpr map key to retrieve metainfo
- Metainfo provides member pointer and type (needed to set member)

Program options: cpp3k

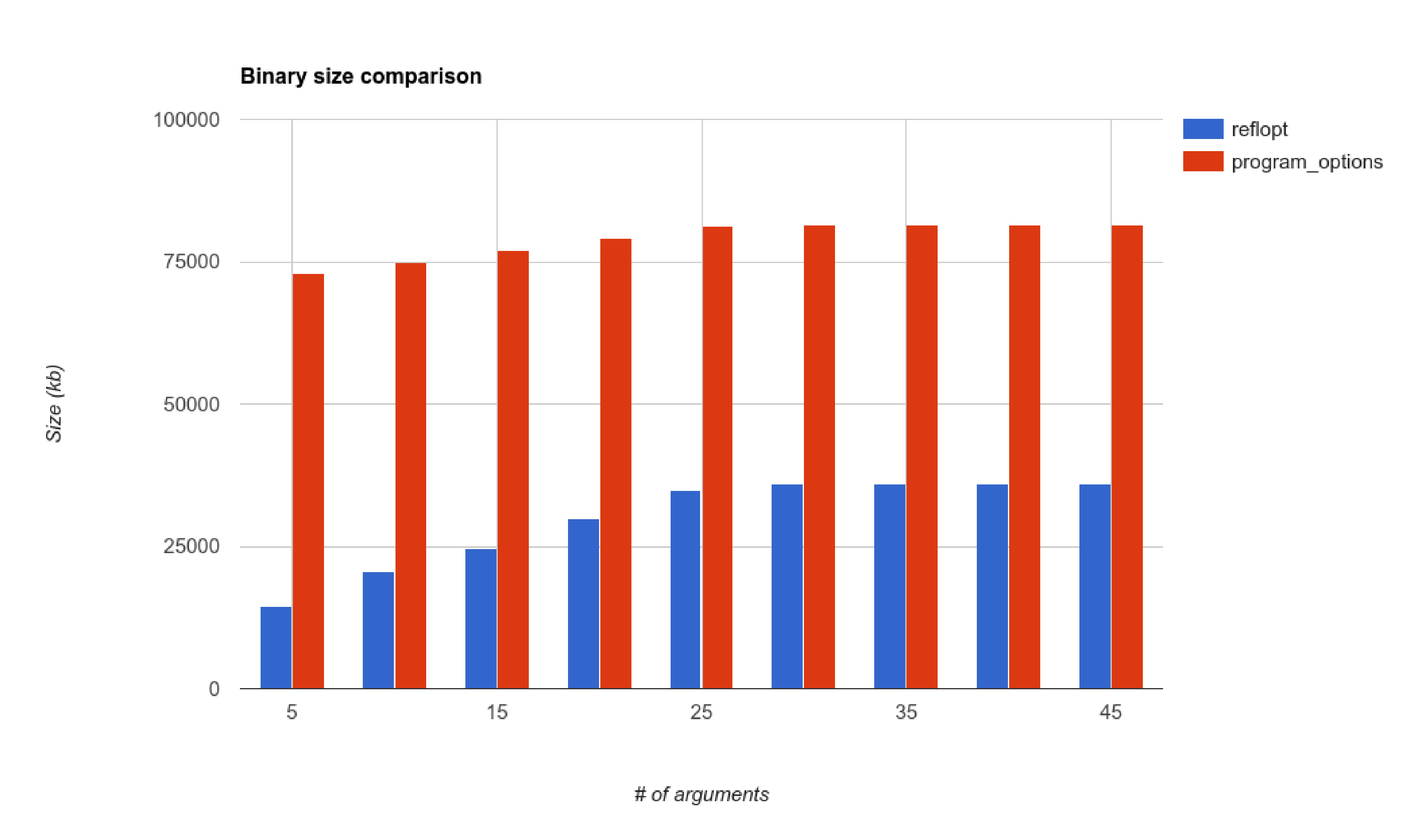
```
template<typename T> struct OptionsMap {
  static constexpr auto prefix_map = hana::fold(
    refl::adapt_to_hana($T.member_variables()),
    hana::make_map(),
    collect_flags
  );
};

/* "adapt_to_hana" internals */
template<typename T, size_t ...I>
constexpr auto adapt_to_hana_helper(const T& t,
    std::index_sequence<I...>&&) {
  return hana::make_tuple(
    cpp3k::meta::v1::cget<I>(t)...
  );
}
```

Parsing and setting members

```
auto set(T& opt, const char* prefix, const char* val) {
  hana::for_each(hana::keys(prefix_map),
    [&options, &prefix, &v](const auto& key) {
      if (runtime_string_compare(key, prefix)) {
        constexpr auto info = hana::at_key(prefix_map,
                                           decltype(key));
        constexpr auto p = info.pointer();
        using M = unreflect_member_t<T, decltype(info)>;
        opt.*p = boost::lexical_cast<M>(val, strlen(val));
      }
    }
  );
}
```

vs. boost::program_options



Observations

- Constexpr strings can be annoying with the current state of the language. We need a standard representation and utilities such as constexpr `strlen` and `strcmp`.
- In order to add metadata such as help strings or custom flags, my ideal syntax is user-defined attributes and reflection on attributes
- Same linear runtime string matching pattern appears. Can we do better?

constexpr string hashing

Idea: exploit our knowledge of the set of all member identifiers at compile time.

Implement a runtime perfect hash from N unique strings to N unique integers (no collisions).

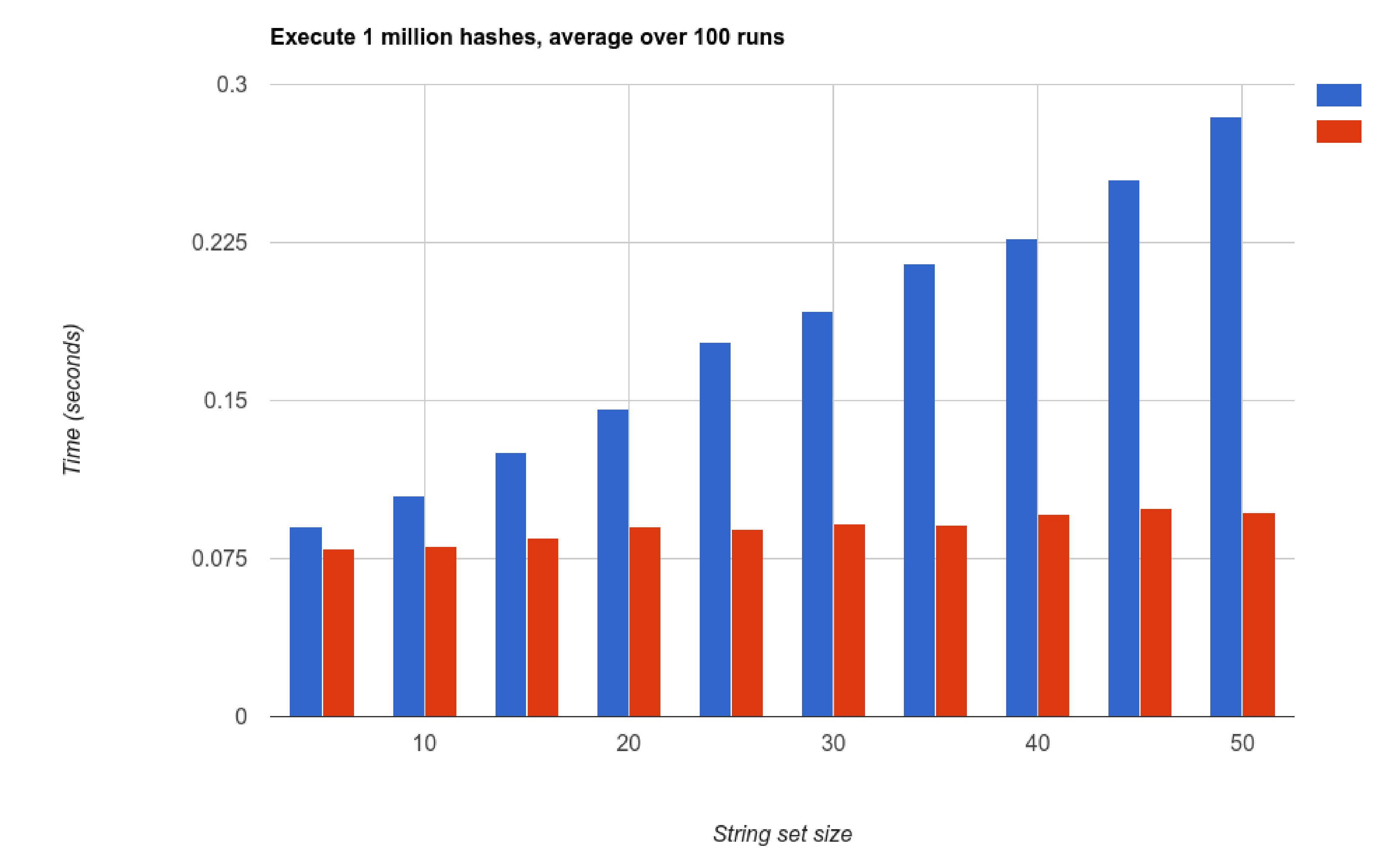
Then match the runtime integer to a compile-time index and a callback.

In our reflection examples, the hash input is a member name, and the hash output is the compile-time index into the struct.

Shameless plug: Petra

- My experimental library for runtime-to-compile time mappings
- Inspired by these reflection examples, but has no dependencies on reflection
- Includes a constexpr string hash (implementation of CHD algorithm)
- https://github.com/jacquelinekay/petra

Results: linear vs CHD hash (-O3)



Caveats/observations

- Number of members in a struct rarely exceeds 10
- Hash is not perfect for all strings from 0 to max-length. Lots of room for improvement
- But, with or without reflection, metaprogramming could become more "practical" with efficient runtime to compile-time mappings

Shifting gears: recent developments

Function reflection: P0670

- `reflexpr` on function names, function declarations, lambdas (non-generic), function parameter names, lambda captures
- `reflexpr` on a name generates an OverloadSet metaobject
- No reference implementation yet

```
void func(int);

using func_overload_m = reflexpr(func);
using func_m = get_element_t<0, get_overloads_t<func_overload_m>>;
using param0_m = get_element_t<0, get_parameters_t<func_m>>;
cout << get_name_v<get_type_t<param0_m>> << '\n'; // prints "int"
```

What's missing?

Introspection is only one piece of the puzzle.

We need the ability to manipulate identifiers at compile-time.

Metaclasses: P0707

Define and re-use a compiler requirements for a set of similar classes

```
$class value {
  constexpr {
    if (find_if(value.functions(),
      [](auto x){ return x.is_default_ctor(); }) != value.functions().end())
    -> { basic_value() = default; }
    /* similar for copy ctor, move ctor, copy assignment, move assignment */
   for (auto f : value.functions()) {
     compiler.require(!f.is_protected() && !f.is_virtual(),
       "a value type must not have a protected or virtual function");
     compiler.require(!f.is_dtor() || !f.is_public()),
       "a value type must have a public destructor");
  }
}
```

Metaclasses: usage

```
value Point { int x;  int y; }
Point p1; // ok, default construction works
Point p2 = p1; // ok, copy construction works

// If we add ordering requirements to value, this works too:
assert (p1 == p1); // ok, == works
assert (p1 >= p2); // ok, >= works
```

Metaclasses enable a lot of uses for reflection!

Conclusion

Reflection is awesome, powerful, and difficult to design for a statically typed language with so many language rules

If you think it's interesting, read the papers, get involved in the SG7 mailing list, and build a reference compilers!

Acknowledgments

This presentation wouldn't exist without:

- Matúš Chochlík, Axel Naumann, and David Sankel for including me in discussions on reflection
- Extra thanks to Matúš for implementing `reflexpr`
- Andrew Sutton for implementing `operator$`
- Louis Dionne for Hana, P0633R0, and string hashing ideas
- Vittorio Romeo for inspiring the program options example, feedback, and moral support

Presentation links

- github.com/jacquelinekay/reflection_experiments
- github.com/jacquelinekay/c++now2017